25/09/2023
Monday

* Remove all adjacents in a string →

I/P = "azxxzy"

azxxzy
a x x y
a y

O/P = "ay"

String s =

a	z	x	x	z	y
0	1	2	3	4	5

String ans =

a	z	x
0	1	2

→ ans.push_back(a);
→ ans.push_back(z);
→ ans.push_back(x);

ans[2] == s[3]
ans.pop_back();

String ans =

a	z
0	1

ans[1] == ans[4]
ans.pop_back();

String ans =

a	y

→ ans.push_back(y);

Approach → If rightmost character of ans string and current character of string s are same, then pop_back from ans string. otherwise push_back the current character of string s in ans string.

__/__/____

Code →

```
#include <bits/stdc++.h>
using namespace std;

string removeDuplicates (string &s){
// new empty ans string for storing required result
   string ans= " ";
   int index= 0;
   while (index< s.length()){
// if ans string ka rightmost character
// and string s ka current character are same
      if (ans.length()>0 && ans[ans.length()-1]== s[index]){
         // pop from ans string
            ans.pop_back();
         }
      else {
         // push in ans string
            ans.push_back (s[index]);
         }
      index++;
   }
return ans;
}
int main() {
    string s = "azxxzy";
    string removeDup = removeDuplicates (s);
    cout<< removeDup;
 }
```

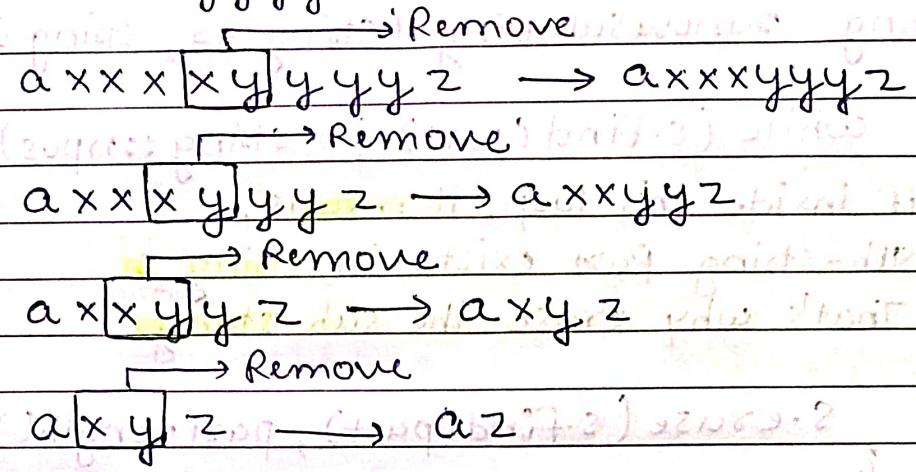
Time complexity → O(n)

Space complexity → O(n) → As we have made a new string, and took extra space.

__/__/____

* **Remove all occurences of substring →**

I/P = axxxxyyyyz



O/P = az

Approach → If sub-string part found in string s, then remove sub-string part and repeat the process.

```
while (s.find(part) != string::npos){

    s.erase(s.find(part), part.length());
}
```

As npos returns true if no matches found else npos returns false.
As sub-string part is present in string s.
So, go inside the loop and erase sub-string part from string s.

__/__/____

Code →

```
#include<bits/stdc++.h>
using namespace std;

string removeSubstring(string &s, string &part){

    while(s.find(part) != string::npos){
// if inside this loop, it means,
// sub-string part exists in string s
// That's why erase the sub-string

        s.erase(s.find(part), part.length());
    }
 return s;
}
int main(){
    string s = "axxxxyyyyz";
    string part = "xy";
    string final = removeSubstring(s, part);
    cout<< final;

}
```

Time complexity = O(n) [T.C. of find() = O(n), T.C. of erase() = O(n)]

Space complexity = O(1)

__/__/___

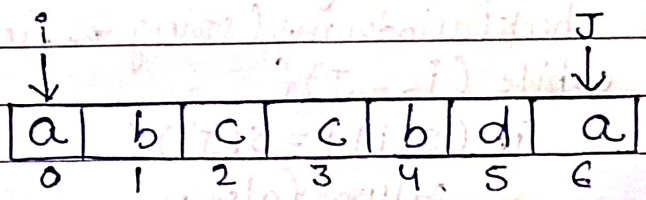
* **Valid or invalid palindrome 2 →**

I/P = "abccbda"

O/P = valid Palindrome.

Condition- Atmost one character removal.

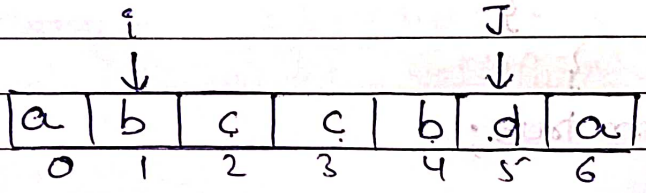
Approach →



s[i] == s[J] → If condition

i++;

J--;



s[i] != s[J] → Else condition

Checking for remaining string (1 character removal only)

(1) ith character removal

ans1 = c c b d → Remaining string is invalid palindrome

(2) Jth character removal

ans2 = b c c b → Remaining string is valid palindrome.

If anyone of the (1) and (2) is true, then return true → return ans1 || ans2

Otherwise, if 0 removals and palindrome is valid, then, simply return true.

__/__/____

Code →

```
#include<bits/stdc++.h>
using namespace std;

// this function checks validity of palindrom for
// the remaining string
    bool checkPalindrome(string &s, int i, int J){
        while (i<=J){
            if (s[i] != s[J]){
                return false;
            else{
                i++;
                J--;
            }
        }
    return true;
}

// this function checks the validity of palindrome
// after at most one removal if needed
    bool validPalindrome (string &s){
        int i=0;
        int J= s.length()-1;
        while (i<J){
            // when s[i]=s[J]
            if (s[i]==s[J]){
                i++;
                J--;
            }else{
            // when s[i] != s[J].
            // 1 removal allowed.
            // check palindrome's validity
            // for remaining string.
```

```
        // when ith character removed
        bool ans1 = checkPalindrome(s, i+1, j);
        // when jth character removed
        bool ans2 = checkPalindrome(s, i, j-1);

// if anyone ans1 or ans2 gives true, return true;
        return ans1 || ans2;
    }
}
return true;
}
int main(){
    string s = "abccbda";
bool isPalindrome = validPalindrome(s);

    if(isPalindrome==1){
        cout<<"valid Palindrome"<<endl;
    }
    else{
        cout<<"invalid Palindrome"<<endl;
    }
}
```

Time complexity → O(n)

Space complexity → O(1)

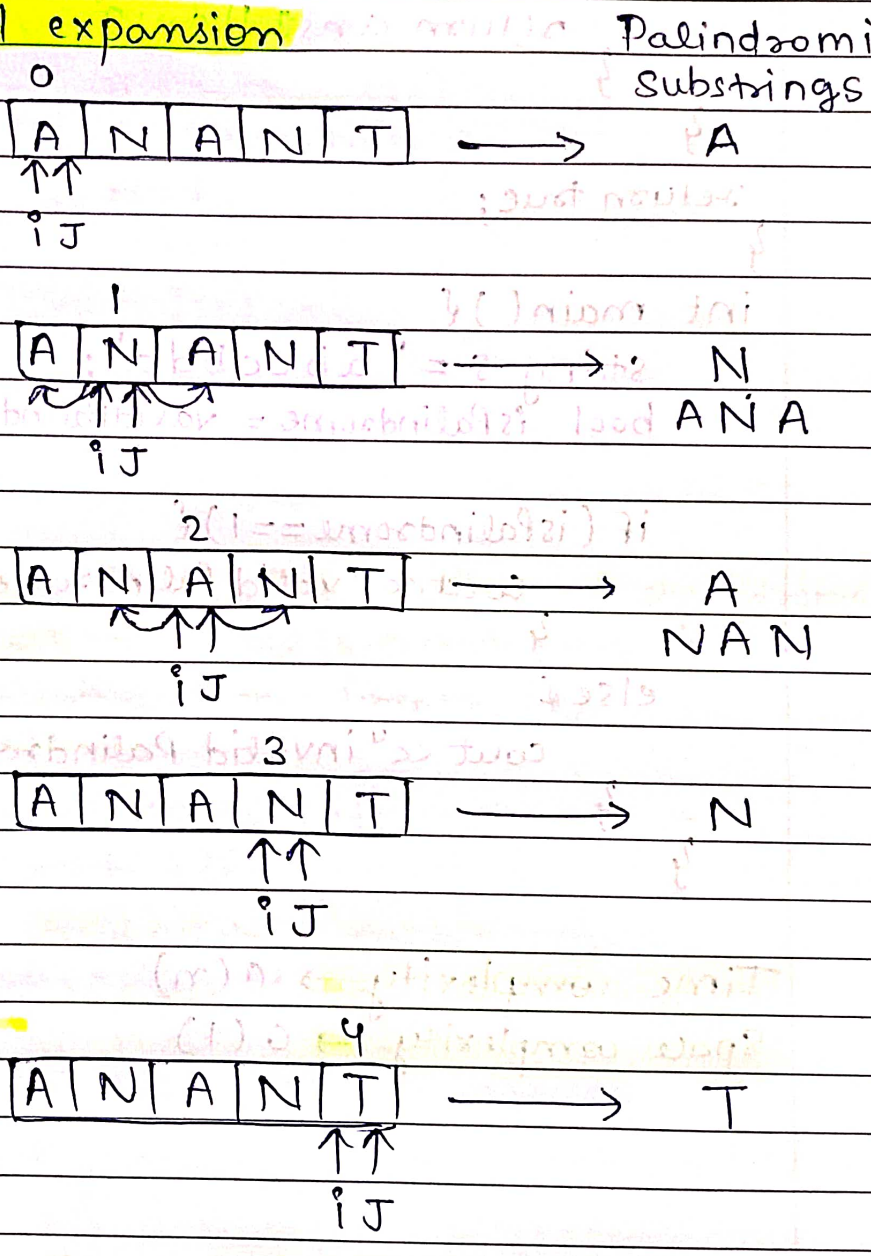
__/__/____

* Palindromic Substrings →

I/P = "ANANT"

O/P = 7

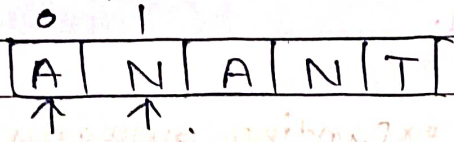
Approach → Odd expansion



OddkaCount = 7

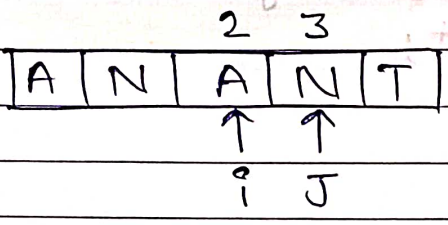
__/__/____

Even expansion — Palindromic substrings

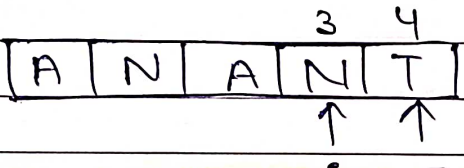


null

null



null



null

evenkaCount = 0

totalCount = oddkaCount + evenkaCount
= 7 + 0
= 7

Assuming every index as the center.

In case of odd length substrings, the substring length expands as 1, 3, 5, 7 - - - so on.

In case of even length substrings, the substring length expands as 2, 4, 6, 8 - - - so on.

Starting from particular index, i and J moves in outward direction i.e by doing i-- and J++.

Code →

__/__/____

```
#include <bits/stdc++.h>
using namespace std;

// this function is for expanding outwards starting from
// center where center indicates every index of string
// and increase the count when s[i] == s[j]
int expand (string &s, int i, int j){
    int count = 0;
    while(i>=0 && j<s.length() && s[i]==s[j]){
        count++;
        i--;
        j++;
    }
    return count;
}

// this function gives the total count of odd and
// even length palindrome substrings exists in string s
int palindromicSubstr (string &s){
    int totalCount = 0;
    // center indicates current index position
    for(int center=0; center<s.length(); center++){
    // odd length palindrome substrings
    int oddkaAns = expand (s, center, center);

    // even length palindrome substrings
    int evenkaAns = expand (s, center, center+1);

    totalCount = totalCount + oddkaAns + evenkaAns
}
return totalCount;
}
```

__/__/____

```
int main(){
    string s = "anant";
    int palindromeCount = palindromicSubstr(s);
    cout << palindromeCount << endl;

}
```

Time complexity → O(n)

Space complexity → O(1)